POLLACK PERIODICA
An International Journal for Engineering and Information Sciences
DOI: 10.1556/606.2017.12.2.1
Vol. 12, No. 2, pp. 3–15 (2017)
www.akademiai.com

A NOVEL PROGRAM SYNTHESIS APPROACH IN TEST DRIVEN SOFTWARE DEVELOPMENT

[1]Endre FERENCZ, [2]Balázs GOLDSCHMIDT

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics, Műegyetem rkp. 3, H-1111 Budapest
Hungary, e-mail: [1]eferencz@iit.bme.hu, [2]balage@iit.bme.hu

Received 30 December 2016; accepted 2 April 2017

Abstract: It is a viable alternative to automatically generate Java source code based on the specification provided by the associated unit tests. This possibility may seem far-fetched in the general case, but after considering the most common restrictions, which are applied nowadays as best practice, it turns out that a significant part of the production code can be generated automatically. The goal is to generate viable implementations, which fulfill the requirements imposed by unit tests. According to the presented vision the modern test frameworks, development guidelines and computational capacities make it possible to reach this goal.

Keywords: Test-driven development, Mocking, Java, Synthesis

1. Introduction

The traditional approach in software development suggests applying test phases after the implementation steps. With the evolution of modern methodologies, these test phases became more and more important in order to reach better software quality. Countless studies confirmed the benefits [1].

Despite the fact that the unit tests are considered so important, they are sometimes neglected, because it is an additional task to keep them up-to-date with the current state of the production code. It is unequivocal that it is possible to generate unit tests from the finished source code [2], but in this way the possible benefits are lost. The most important value comes from the fact that the developer has to think through the requirements from a different point of view.

Hereby, the other way around is promoted: make it possible to generate the production code from the test code by adhering to a few basic guidelines, most importantly to mock the dependencies.

Most of the time there is an unlimited number of possible solutions, out of which the suitable ones can be selected by using heuristics. Showing possible implementations might significantly speed up the development process.

This approach promises other possible advantages, which include full test coverage, mutation-based testing techniques, better code quality, another form of documentation and code re-use.

2. Related work

The concept of automatic programming has a long history. In 1954 the term was used to describe (Fortran) compilers [3]. In the early ages of computer programming, even a simple compiler could improve significantly the productivity of a software engineer. The currently available hardware makes it possible to use high level programming languages, which abstract away the prosaic concerns of software development, like garbage collection or pointer handling.

Automatic programming aims to provide a higher-level approach than the one which is currently available for the programmer [4]. At present, an extended form of this concept is used. It covers the synthesis of an algorithm based on various models. It might provide a correctness proof for the algorithm. The main target of using these formal methods for program synthesis is the embedded systems and safety-critical systems [5]. It is expensive and time consuming to formalize a specification, but it has huge benefits. Upon using a certified tool for program synthesis, the source code development and verification happens almost instantly.

There are several techniques for program synthesis. Traditionally they use a form of theorem proving [6]. In these cases, the input of the process is a specification in a standard equation form. Another approach is to implement the specification as a simpler, but less efficient program [7]. These have the advantage of completeness of specification and proven verification, but these specifications are often hard to write and difficult to check with an automated formal verification technique.

The characteristics of the domain (of the software under development) in a significant part of the cases make it unfeasible to use formal methods [8]. Sometimes it is worth to decompose the task based on these characteristics. There are several agile methodologies, which aim to embrace formal methods in a modern software development project [9], [10].

Automatic program synthesis has many practical applications [11], [12]: automating repetitive programming tasks [13], reaching optimal code sequences [7], unfolding high-level specifications [14], [15] or reverse-engineering obfuscated software components [11]. These are very specific goals, which may be helpful in the targeted situation, but they do not provide aid in a software development process in an unobtrusive manner.

It is not uncommon to use a component-based approach in program synthesis [11]. Modern enterprise applications frequently use the abstraction level of components [16] in order to reach the appropriate level of maintainability and flexibility.

Most of the program synthesis tools aim to handle a limited part of the codebase: a few instructions, a method or an algorithm. For example, the peephole optimizers [17] work with a limited number of instructions. There are approaches, which target a top-down strategy [18]. These create a decomposition of the initial specification into a hierarchy of specifications for sub-problems.

3. Restrictions

A widespread solution for managing complexity [19] is to apply strict regulations regarding the size of different source entities. For example, classes should have one responsibility (one reason to change) or functions should do one thing, [20]. If the software engineers follow these conventions, methods with just a few lines of code become quite typical. In these cases, the number of conceptually similar methods is higher and it makes the pattern recognition algorithms much more feasible. This research is narrowed to methods with a few loops (maximum two) and a limited number of lines (maximum 20).

The strong restriction of statelessness was chosen. Stateless components are very common because of the simplicity and inherent thread safety. This restriction might be bypassed by treating the actual state as a parameter. These stateless components operate on data objects, which contain only minimal behavior: getters-setters, hash code calculation, equality test and possible serialization algorithm.

Another important presumption is that the tests should be as specific as possible. The output of the operations should be verified extensively, preferably with equality assertions.

According to this methodology, the units (the classes) are tested in isolation without dependencies. This is achievable by using a mocking framework [2]. The used test doubles may contain the minimal amount of logic, which is needed for proper testing, but the boundaries between the components under development should be specific in regards of the parameters and return values.

4. Importance of mocking

In everyday software development - upon unit testing - it is appropriate to use a mocking framework in order to test a class in isolation, independently from its dependencies. Generally, this approach is inevitable in situations like: unavailability of dependencies (e.g. unfinished software components), high cost of invoking/testing dependencies (e.g. web service, database connection), strict isolated testing (e.g. avoid the effects of deficient dependencies), nondeterministic behavior (e.g. random), special interactions (e.g. test callbacks) [2].

In the examples from *Listing 1* and *Listing 2* the strong relationship between the production code and the test code is demonstrated. The behavior of the dependencies is

specified in *Listing 2*. These mocked methods represent valuable information for the process of source code generation, because - when used in this specific form - show a snapshot of their parameters from the data flow and specify additional input data for the subsequent operations. In this way the additional complexity of the dependencies is completely avoided and the flow of the executed business logic is specified with more details.

Listing 1

Example of a business method with external dependencies (*productStore* and *categoryStore*)

```
BigDecimal getVatFraction(Long productId) {
   Product product = productStore.loadProduct(productId);
   Category category =
      categoryStore.loadCategory(product.getCategoryId());
   return category.getVatFraction();
}
```

Listing 2

Test code for isolated testing of business method from *Listing 1*

```
void testGetVatFraction() {
   // Setup
   Long productId = 73L;
   Long categoryId = 93L;
   BigDecimal vatFraction = new BigDecimal("0.27");
   Product dummyProduct = new Product();
   Category dummyCategory = new Category();
   dummyProduct.setCategoryId(categoryId);
   dummyCategory.setVatFraction(vatFraction);
   when(productStore.loadProduct(productId))
      .thenReturn(dummyProduct);
   when(categoryStore.loadCategory(categoryId))
      .thenReturn(dummyCategory);
   // Exercise
   BigDecimal result = vatCalculator
      .getVatFraction(productId);
   // Verify
   assertSame(vatFraction, result);
}
```

The previously described code snippet represents a straightforward situation, because the test method abides by the previously described guidelines. To generate the source (*Listing 1*) based on the test (*Listing 2*), first the data dependencies between the return value, the two mocked methods and the input parameters have to be analyzed.

The analysis in this case results in a straightforward process, because the input-output pairs can be clearly identified. The process involves a basic transformation in the form of calling getter methods of data classes (*Product* and *Category*). The illustration of the result (*Fig. 1*) for the source from *Listing 1* shows the basic data flow of the input parameters and how they are derived from the available variables. It demonstrates a clear path of calculations.

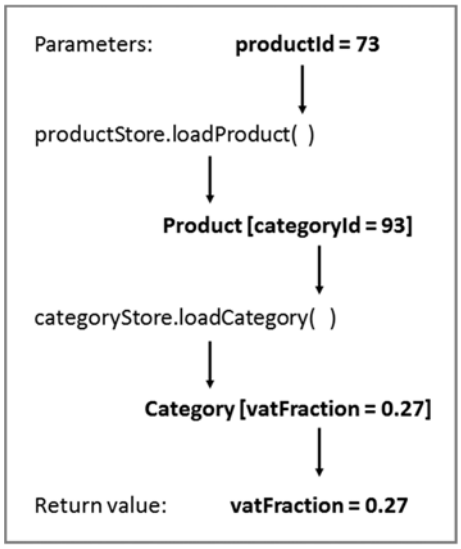

Fig. 1. Data dependency analysis

Some pragmatic cases manifest algorithms and transformations that are more sophisticated. These might be addressed by using different techniques, but even the coverage of the simplest situations - like the example before - results in a great benefit in day-to-day development tasks.

5. The source generation process

Three different approaches are proposed to generate the source code from the test methods. These use the following information, which are available from the test methods:

- Signature of the method;
- Input and assertions of the output test data;
- Parameters (optional) and the results of the invoked external dependency methods from mocks;
- Number of invocations to the different external dependency methods.

5.1. Naive approach

The naive approach is suitable to decide whether the provided test data and behavioral information is theoretically possible to be fulfilled with a stateless method, tested in isolation. An example for such unrealizable situation is expecting different outcomes from the same input data and environment, because these un-deterministic fragments should be isolated for reliable unit testing.

The naive approach implements separate branches (if statements) based upon the input data, then it invokes the external dependency methods with the predefined parameters and finally decides the result of the method.

Irresolvable situations might appear if the algorithm cannot decide between the different external dependency method calls, specified by the different test cases. This situation is only possible if the input data specified by two test cases are identical, but these test cases provide incompatible external method calls.

After the successful invocation of external methods, the input data expands with the results of these calls and then the same decision has to be made, but now the goal is to determine the result of the business method.

To demonstrate the naive approach more pragmatically a simple method is targeted (without external dependencies) from an open source library. The source code with minor refactoring is visible in *Listing 3*. This method might be tested thoroughly by using the test cases from *Table I*.

Listing 3

Moderately complex business logic from the *PagedListHolder* class of the *Spring framework*

```
int getPageCount(int nrOfElements, int pageSize) {
   float nrOfPages = (float) nrOfElements / pageSize;
   boolean plusOne =
      nrOfPages > (int) nrOfPages || nrOfPages == 0.0;
   return (int) (plusOne ? nrOfPages + 1 : nrOfPages);
}
```

Table I

Test data used for the sample code from *Listing 3*

nrOfElements	pageSize	result
1	1	1
5	3	2
13	2	7
8	7	2
0	5	1

This mechanism would generate the control flow from *Fig. 2*. For each tested input the output is available, so a branch is generated for each input pair. The unspecified

inputs in this case can only be covered with an exception, because the other results cannot be predicted.

5.2. Constructive approach

The Constructive approach is a form of program synthesis where the specification is not a precise formula, just input and output data pairs. Of course, this gives space to multiple or even an unlimited number of possible method implementations. According to the herby presented vision, this is even desirable, because showing multiple - significantly different - solutions to the same problem (which give diverse results for the unspecified inputs) demonstrates that the provided test cases does not provide enough certainty.

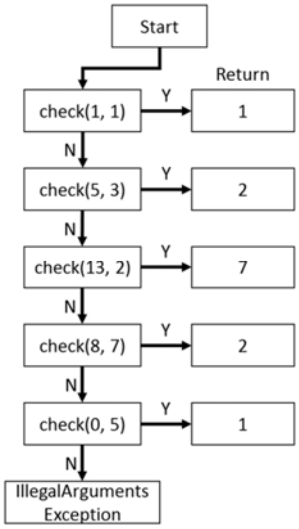

Fig. 2. The generated control flow of the naive approach

The implementation of this approach (*Listing 4*) starts from the input parameters and then traces the possible calculations to the output values. Possible calculation steps may come from: data transformations, mocked methods and control structures (branch, loop). The algorithm generates every possible output value given the inputs and the calculation steps. If the output values match the expected test results, the path method retraces the executed calculation steps for reproduction. The alternatives collection might be used to demonstrate alternative implementations for the developer, because it stores alternative ways to reach some of the values.

It is possible to manage the cases where the data output of the targeted method is unknown. The different calls to mocked methods and their parameters are considered to be assertions, which should be fulfilled as well.

The algorithmic complexity may get out of hand upon chaining multiple transformations and applying multiple control structures. In practice the number of

possible transformations is limited by the data type in use, so an extensive search is possible in this manner. Generally, the synthesis is hindered by the control statements, which increase the method complexity.

There are multiple metrics [21], which are targeted to measure the complexity of a method. The most notable ones are the cyclomatic complexity [22] and the NPath complexity [23]. Generally, these are limited on a software development project to ensure the maintainability, so - from an algorithmic complexity point of view – these are considered as small constant values.

Listing 4

Algorithm of the constructive approach

```
Input:
 tests: test data (input-output pairs)
 mockCalls: test data (input-output pairs)
 operations: allowed operations

Algorithm:
 interimStates = empty set;
 interimStates.add(input(tests));
 alternatives = empty set;
 while (not interimStates contain output(tests)) {
  for (each (op, s[]) from
   (operations union mockCalls, interimStates)) {
   if (typeMatch(op, s)) {
    newState = (op(s), s, op);
    if (not interimStates contains newState) {
     interimStates.add(newState)
    } else {
     alternatives.add(newState)
    }
   }
  }
 }
 match = find(interimStates, output(tests))
 code = path(match)
```

Branches can also be synthesized if their usage implies shorter execution paths. A sufficient amount of test data is needed, because branches can only be generated if complete branch coverage is achieved.

The most insecure part of the process is the consolidation of conditional expressions. Generally, it is recommended to use boundary testing in order to ensure the best output of the classification process.

Mechanism for solving the general case of re-engineering a loop has not been found. Some common cases can be easily detected (for example an iteration over a collection), but even a simple prime check seems impossible to reproduce using automatic

programming without further hints. The detection of these types of algorithms can be implemented explicitly in the source generator engine.

Multiple optimizations are applied in order to reduce the number of combinations in the program synthesis. The primary goal is to produce one possible implementation, but showing multiple possibilities is also desired.

It is common that two or more simple calculation fragments result in the same outcome when used with the same input test data. In this case, only one should be kept for further work, the other ones just make it possible to produce more than one solution.

Generally, the type of the input data imposes strict constraints upon the possible transformations, which makes it possible to retrace longer calculation flows. Other data types might provide a multitude of possible operations, which makes unfeasible the synthesis process. In such cases, it is advisable to provide hints to the engine in order to avoid unnecessary calculations with irrelevant steps.

In general, the best approach for effective synthesis is to design short methods, which are only meant to do one thing. Multiple well-known patterns suggest such solution; the most notable one is the *composed method pattern* [24] from Kent Beck.

5.3. Abstract search based approach

Software engineers re-implement well-known algorithms or code snippets frequently. An example for this situation is presented in *Listing 3*. Certainly this *page-count* method appears in almost every user interface framework. On one hand despite of this certainty it might seem strange to use the common way of dealing with duplication and to provide a library with these methods. On the other hand, if there were a library with this method, it would be time consuming to find it.

The abstract search based approach suggests a solution to this problem. The implementations are stored and indexed in order to make them available for source code synthesis. Developers may search between these predefined implementations by using the provided information from the tests.

Several code duplication detection methods are already available for use [25]. Generally, they use some kind of abstraction in order to reveal repetitions, which are not line-by-line equivalent, but logically identical. These techniques are good candidates for further improvement.

During the abstraction process, the reproducible parts of the source code should be suppressed. For example, in case of a parameter, which is a simple data class, it is irrelevant what the concrete type is, the used primitive data types and their constraints should be kept only.

The methods from external dependencies should be omitted too, because according to the previously assumed restrictions, they will be mocked out during the testing phase.

Finally, the most relevant parts are the control structures and most importantly the loops, because they cannot be reproduced based upon the tests.

The number of stored algorithms grows quickly, so these should be indexed using a B+ tree, which guarantees logarithmic time complexity upon query operations. The actual indexed values should consist of the test data. As new searches with appropriate data types are started, these indexes should be supplemented with the newly tested values.

5.4. Effectiveness of the abstract search based approach

The abstract search based approach is effective for frequently used well known algorithms. The most common challenges are most likely to be even covered by multiple algorithms. Generating the business method demonstrated in *Listing 3* with the constructive approach is a computationally intensive task, because of the long and complex transformation chain demonstrated in *Fig. 3*. In this figure the nodes represent calculation steps, the arrows show where the input values come from.

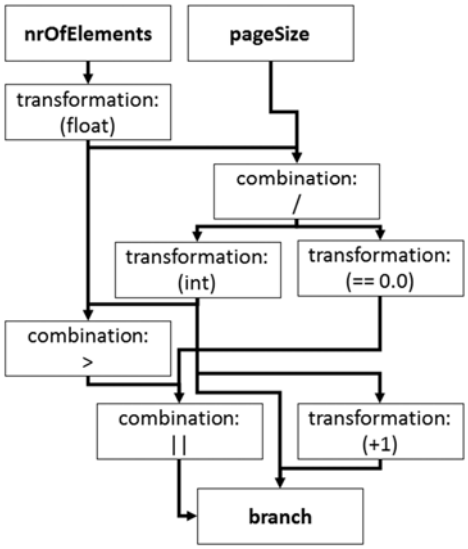

Fig. 3. The transformation chain of the algorithm from *Listing 3*

The longest transformation chain in the process has five transformations/ combinations and a branch. Upon producing long transformation chains the size of the search space grows exponentially. The implemented tool managed to reproduce similarly complex algorithms in a short time frame (less than one minute) on commodity hardware by limiting the number of possible instructions to the ones, which appear in this calculation.

Even with a large number of stored algorithms and test data, the lookup is generally much faster than the constructive approach, because it has a logarithmic complexity instead of the exponential complexity of the constructive approach.

6. Advantages

6.1. Benefits for software development

A source code generator cannot substitute the work of the software engineer, because the generated code has to be interpreted and reviewed to ensure that enough test

cases are provided to reach the goals. This tool may be useful to present alternative implementations, which might be the base of the final source code.

A recurring issue in everyday software development is to obey the coding guidelines, apply the usual patterns in the source code. Automatic program synthesis may contribute significantly to reach these goals, because these aspects might be easily preconfigured.

In order to provide these benefits a test driven development technique should be in use.

6.2. Benefits for unit testing

Upon testing the algorithm from *Listing 3* with the test data from *Table I* code coverage tools report 100% instruction and branch coverage.

The constructive approach was used to reproduce the same logic from the test data. The number of possible instructions was limited to the few relevant ones: type conversions between integer and float, division, zero equality check, or Boolean operator and greater than relational operator.

The shortest solution provided by the generator tool (*Listing 5*) was very similar, but with a minor difference. The condition of the branch has changed, so the transformation chain is shorter by one transformation.

Listing 5

Reproduced version of the method from *Listing 3*

```
int getPageCount(int nrOfElements, int pageSize) {
   float nrOfPages = (float) nrOfElements / pageSize;
   boolean plusOne =
      nrOfElements > pageSize || nrOfElements == 0;
   return (int) (plusOne ? nrOfPages + 1 : nrOfPages);
}
```

This is a mutation of the original code, which is not covered by the provided unit tests. Adding the (12, 2) test to the test suite provides the necessary test coverage for the tool to generate the same logic as the original method.

Mutation testing tools contain a predefined set of mutators and the goal is to produce non-equivalent mutations. The proposed approach is to generate test-equivalent, but shorter mutations. In this way, those parts of the source code can be highlighted, which are not justified by the associated unit tests.

7. Conclusions

In this article a novel approach in test driven software development was presented. It started with describing the state of the art techniques in enterprise software development, which are essential for the feasibility of the approach. Experimental

source code generation techniques were described and demonstrated on common code snippets. Finally, the possible benefits and usages of the tool were discussed.

Currently the project is in a prototyping state, which means that the usage is cumbersome, it is time consuming to analyze a larger codebase. The next goal is to measure the amount of source code, which could have been generated with this methodology on a modern software development project.

The long-term goal is to increase this coverage and to present a tool, which can automatically process the provided unit tests in order to suggest viable implementations based on its configuration.

References

[1] Bertolino A. Software testing research, Achievements, challenges, dreams, *Proceedings on Future of Software Engineering, FOSE '07,* IEEE Computer Society Washington, DC, USA, 23-25 May 2007, pp. 85−103.

[2] Saff D., Artzi S., Perkins J. H., Ernst M. D. Automatic test factoring for Java, *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering, ASE '05*, Long Beach, CA, USA, 7-11 November 2005, pp. 114−123.

[3] Balzer R. A 15 year perspective on automatic programming, *IEEE Trans. Software Eng.* Vol. SE-11, No. 11, 1985, pp. 1257−268.

[4] Parnas D. L. Software aspects of strategic defense systems, *Communications of the ACM*, Vol. 28, No. 12, 1985, pp. 1326−1335.

[5] Woodcock J., Larsen P. G., Bicarregui J., Fitzgerald J. Formal methods: practice and experience, ACM Computing Surveys, Vol. 41, No. 4, 2009, pp. 1−40.

[6] Madden P., Hesketh J., Green I., Bundy A. A general technique for automatically generating efficient programs through the use of proof planning, *Proceedings of LOPSTR 93, International Workshop on Logic Program Synthesis and Transformation*, Louvain-la-Neuve, Belgium, 7–9 July 1993, pp. 64−66.

[7] Massalin H. Superoptimizer, a look at the smallest program, *Proceedings of the Second International Conference on Architectual Support for Programming Languages and Operating Systems*, *ASPLOS II,* Palo Alto, California, USA, 1987, pp. 122−126.

[8] Liu S., Adams R. Limitations of formal methods and an approach to improvement, *Proceedings 1995 Asia Pacific Software Engineering Conference*, 6-9 December 1995, pp. 498−507.

[9] Shafiq S., Minhas N. M. Integrating formal methods in XP, A conceptual solution, *Journal of Software Engineering and Applications*, Vol. 7, No. 4, 2014, pp. 299−310.

[10] Marciniak R. Role of new IT solutions in the future of shared service model. *Pollack Periodica,* Vol. 8, No. 2, 2013, pp. 187−194.

[11] Jha S., Gulwani S., Seshia S. A., Tiwari A. Oracle-guided component-based program synthesis, *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - ICSE '10*, Cape Town, South Africa, 2-8 May 2010, Vol. 1, pp. 215−224.

[12] Cao X., Kui Y., Qiaoyu X.. A deep study to reuse of processing instructions in automatic programming techniques, *2016 IEEE International Conference on Mechatronics and Automation,* 7-10 August 2016.

[13] Lau T., Domingos P., Weld D. S. Version space algebra and its application to programming by demonstration, *Proceedings of the Seventeenth International Conference on Machine Learning*, ICML '00, 29 June - 2 July 2000, pp. 527–534.

[14] Solar-Lezama A., Tancau L., Bodik R., Seshia S., Saraswat V. Combinatorial sketching for finite programs, *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems,* ASPLOS XII, San Jose, California, USA, 21-25 October 2006, pp. 404−415.
[15] Kilián I. When the snake bites its own tail... Messing up and solving meta-levels in informatics and in everyday problems, *Pollack Periodica*, Vol. 5, No. 2, 2010, pp. 69−79.
[16] Herzum P., Sims O. *Business component factory, A comprehensive overview of component-based development for the enterprise,* New York, John Wiley, 2000.
[17] Bansal S., Aiken A. Automatic generation of peephole super-optimizers, *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, California, USA, 21-25 October 2006, ACM SIGARCH Computer Architecture News, Vol. 34, No. 5, 2006, pp. 394−4023.
[18] Smith D. R. Top-down synthesis of divide-and-conquer algorithms, *Artificial Intelligence,* Vol. 27. No. 1, 1985, pp. 43−96.
[19] Vasileva A., Schmedding D. How to improve code quality by measurement and refactoring, *2016 10th International Conference on the Quality of Information and Communications Technology,* 6-9 September 2016.
[20] Martin R. C. *Clean code, A handbook of agile software craftsmanship,* Upper Saddle River, NY, Prentice Hall, 2009.
[21] Wallace L. G., Sheetz S. D. The adoption of software measures: A technology acceptance model (TAM) perspective, *Information & Management*, Vol. 51, No. 2, 2014, pp. 249−259.
[22] McCabe T. J. A complexity measure, *IEEE Trans. Software Eng*. Vol. SE-2, No. 4, 1976, pp. 308−320.
[23] Nejmeh B. A. NPATH: A measure of execution path complexity and its applications, *Communications of the ACM,* Vol. 31, No. 2, 1988, pp. 188−200.
[24] Beck K. *Implementation patterns,* Upper Saddle River, NY, Addison-Wesley, 2008.
[25] Roy C. K., Cordy J. R., Koschke R. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach, *Science of Computer Programming,* Vol. 74, No. 7, 2009, pp. 470−495.